Programmation 1, leçon 1

Jean Goubault-Larrecq
ENS Cachan
goubault@lsv.ens-cachan.fr

19 septembre 2014

1 Une brève introduction aux langages de programmation

Il existe de nombreux langages de programmation. En fait, en comptant les langages d'usage général, les langages de script, les langages dédiés à une application, ..., le nombre de langages existants se compte probablement en dizaines de milliers. Au jour du 16 septembre 2013, la page Wikipedia `http://fr.wikipedia.org/wiki/Liste_des_langages_de_programmation` en liste 604.

Aujourd'hui, d'après la page Wikipedia en anglais `http://en.wikipedia.org/wiki/Programming_language`, laquelle cite une consultation en date du 03 décembre 2010 du site `langpop.com`, les langages les plus populaires sont, en ordre alphabétique : C, C++, C#, Java, JavaScript, Perl, PHP, Python, Ruby, et SQL.

Ce sont pour les plupart des langages *impératifs*, c'est-à-dire où l'exécution procède par changements d'états successifs. (Certains ont diverses particularités en plus, par exemple C++, C#, Java, JavaScript, Python et Ruby sont aussi orientés objet.)

Par exemple, la factorielle (un exemple que je reprendrai jusqu'à la nausée dans la suite de ce cours) s'écrit en C :

```
int fact (int n)
{
  int resultat;
  int i;

  resultat = 1;
  for (i=1; i<=n; i++)
    resultat = resultat * i;
  return resultat;
}
```

ce qui se lit informellement : "pour calculer la factorielle de n, mettre le résultat à 1, puis le multiplier successivement par 1, par 2, ..., par n".

À l'opposé, les langages *déclaratifs* tentent de décrire ce qu'on souhaite calculer sans décrire comment — ou, sans être aussi extrémiste, insistent sur le quoi au détriment du comment. Par exemple, la factorielle en un langage *fonctionnel* comme CaML s'écrira :

```
let rec fact n =
        if n=0
           then 1
        else n * fact (n-1);;
```

ce qui correspond bien à la notation mathématique définissant la factorielle :

$$n! = \begin{cases} 0 & \text{si } n = 0 \\ n.(n-1)! & \text{si } n \neq 0 \end{cases}$$

Un autre exemple est donné par les langages *logiques*, dont le représentant typique est Prolog, où l'on écrirait typiquement :

```
fact(0,1).
fact(N+1,Y) :- fact(N,Z), Y=(N+1)*Z.
```

(J'ai en réalité un peu triché dans l'écriture de ce programme, pour des raisons de lisibilité.) Ceci définit une relation `fact` entre deux entiers, telle que `fact`(m, n) est censé être vrai si $n = m!$, et définie par le fait que $1 = 0!$, et $Y = (N + 1)!$ s'il existe $Z = N!$ tel que $Y = (N + 1)Z$. Je n'insisterai pas sur les langages logiques, que vous étudierez dans d'autres cours à l'ENS Cachan.

Le but de ce cours n'est pas spécifiquement de vous apprendre à programmer dans tel ou tel langage de programmation, mais de vous apprendre quelques concepts de base communs à pratiquement tous les langages de programmation.

Un concept central est celui de *sémantique*. Plutôt que de dire de quoi il s'agit tout de suite, examinons les questions suivantes :

– Je vous donne un programme, par exemple le programme `fact` écrit en C ou en CaML comme ci-dessus. Que fait ce programme ?
  Vous pouvez vous laisser guider par une intuition plus ou moins vague de ce que fait chaque instruction du programme. Mais ceci ne mène pas toujours loin. Par exemple, sachant que, en C, x++ ajoute un au contenu de la variable x et retourne la valeur qu'avait x avant, que fait l'instruction suivante ?
  `x = x++;`
  Vous pouvez tester ce programme, et par exemple sur ma machine, ceci ajoute juste 1 à `x`. Il est probable que, si vous avez un compilateur C qui fonctionne comme le mien, vous soyez convaincus que c'est effectivement ce que doit faire cette instruction. (J'expliquerai la notion de compilateur un peu plus loin.) Mais un jour vous le ferez peut-être tourner sur une autre machine, ou avec un autre compilateur, et l'instruction ci-dessus ne changera pas la valeur de `x`. La raison en est que `x = x++;` demande essentiellement à effectuer trois opérations :

  1. lire la valeur de `x` avant, appelons-la x_0 ;

2. réinscrire $x_0 + 1$ dans la variable `x` (l'effet de `x++`) ;
3. mettre x_0 (la valeur de `x++`) dans `x`.

Or les deux dernières entrent en conflit : on n'obtient pas le même résultat selon qu'on exécute 2 avant 3 ou 3 avant 2, et la norme du langage C ne précise aucun ordre entre ces actions. Plus surprenant, il n'est pas exclu qu'un compilateur décide de traduire cette instruction sous forme d'instructions assembleur qui mettent essentiellement *n'importe quoi* dans `x`.

Savoir exactement les effets possibles d'une instruction donnée, c'est en connaître la sémantique.

– J'ai écrit deux versions de la factorielle, en C et en CaML, ci-dessus. Comment puis-je être sûr qu'ils calculent la même chose ?

  Évidemment, j'ai tout fait pour, donc ils devraient calculer la même chose... mais j'ai pu me tromper (il peut y avoir des *bogues*). Comment sais-je que je ne me suis pas trompé, ou comment puis-je trouver les bogues ? Il y a plusieurs approches : utiliser un *débogueur*, ou effectuer des campagnes de tests, permettent de détecter ou de comprendre des erreurs, du moins si ces erreurs se manifestent suffisamment fréquemment et de façon suffisamment reproductible.

  On peut aussi tenter de démontrer un théorème d'équivalence des deux programmes. Ceci revient à démontrer que les deux programmes ont la *même sémantique*, ou dans les cas plus complexes, ont des sémantiques reliées par des relations ayant de bonnes propriétés (relations d'abstraction, de raffinement, bisimulations ; ce ne sera pas le programme de ce cours).

  Pour ceci, on aura besoin de définir *mathématiquement* les diverses sémantiques de nos langages de programmation.

– J'ai parlé un peu plus haut de compilateurs (vers le langage assembleur). Un compilateur est un programme, qui prend le texte source d'un programme en entrée (par exemple en C ou en CaML), et le traduit en un autre langage (typiquement le langage machine, ou assembleur). Une sémantique, c'est fondamentalement pareil : c'est une fonction prenant en entrée un texte source et retournant un objet mathématique.

  Si l'on arrive à considérer les programmes (C, CaML, assembleur) comme des objets mathématiques, les compilateurs ne seront rien d'autre que des fonctions de sémantique particulières. L'avantage de cette vision sera en particulier que l'on pourra démontrer qu'un compilateur est correct par rapport à une sémantique : ceci reviendra à montrer que deux sémantiques sont équivalentes (typiquement, que ce sont deux fonctions égales).

Ce cours abordera donc des extrêmes assez éloignés :

– D'un côté nous aurons un peu de mathématiques : sémantiques dénotationnelles, opérationnelles ; théorie de l'ordre et théorèmes de points fixes, théorie de la démonstration et récurrences structurelles.
– De l'autre nous aurons à voir quelques considérations très pratiques : pour comprendre la sémantique de l'assembleur, et même certaines particularités de C comme les pointeurs, il faudra comprendre comment fonctionne un ordinateur : processeur, mémoire, et notions d'adressage notamment. Le cours d'architecture matérielle de Stefan Schwoon vous sera ici d'une aide précieuse.

Avant de passer à la suite, considérez le programme suivant en C, si vous comprenez déjà le C :

```
void merge (int *l1, int n1,
            int *l2, int n2,
            int *res) {
  while (n1!=0 && n2!=0) {
    if (*l1 < *l2)
       { *res++ = *l1++; n1--; }
    else { *res++ = *l2++; n2--; }
  }
  while (n1--!=0) *res++ = *l1++;
  while (n2--!=0) *res++ = *l2++;
}
```

```
void sort1 (int *l, int n,
            int *res) {
  int k;

  if (n==0) return;
  if (n==1) { *res = *l; return; }
  k = n / 2;
  sort1 (l, k, res);
  sort1 (l+k, n-k, res+k);
  merge (res, k, res+k, n-k, l);
}
```

```
void sort (int *l, int n) {
  int *aux = (int *) malloc (n * sizeof (int));
  sort1 (l, n, aux);
  free (aux);
}
```

Il s'agit d'un programme de tri d'un tableau `l` d'entiers, de longueur `n`, par la méthode dite de *tri par fusion*. Ce programme contient un gros bogue : lequel ? Vous pouvez le trouver par test et déboguage, ou en raisonnant formellement. Dans tous les cas, je ne pense pas que vous voyiez lequel il est en moins de cinq minutes. Ceci nous mène à l'idée qu'on aimerait disposer de méthodes automatiques de preuve de programme. La sémantique sera indispensable pour cela, mais nous ne traiterons pas ici de cet aspect. La logique de Hoare, par exemple, sera vue dans le cours d'algorithmique de Paul Gastin.

Un autre exemple que je prendrai dans ce cours est le petit programme de la figure 1.

Tapez-le dans un fichier que vous nommerez `cat.c`. Il s'agit d'un autre exemple que je réutiliserai jusqu'à la nausée. Il s'agit du texte source de l'utilitaire `cat` d'Unix. (Du moins, d'une version naïve, mais qui fonctionne.) Ce programme est censé s'utiliser en tapant sous le *shell* (interprète de commandes Unix), par exemple :

```
cat a b
```

ce qui va afficher le contenu du fichier de nom `a`, suivi du contenu du fichier de nom `b`. (Vous aurez pris soin de créer deux fichiers nommés `a` et `b` d'abord, bien sûr.) En général, `cat` suivi de noms de fichiers f_1, …, f_n, va afficher les contenus des fichiers f_1, …, f_n dans l'ordre ; ceci les con`cat`ène.

Si, au lieu d'utiliser l'utilitaire Unix `cat`, vous essayez d'utiliser le programme `cat.c` ci-dessus, par exemple en tapant :

```
cat.c a b
```

```
#include <stdio.h>

main (int argc, char *argv[])
{
  int i, c;

  for (i=1; i<argc; i++)
    {
      FILE *f;

      f = fopen (argv[i], "r");
      while ((c = fgetc (f))!=EOF)
        fputc (c, stdout);
      fclose (f);
    }
  fflush (stdout);
  exit (0);
}
```

FIGURE 1 – Un programme de concaténation en C

vous verrez que cela ne fonctionne pas. C'est parce que le *processeur* de la machine, c'est-à-dire le circuit intégré qui fait tous les calculs dans la machine en face de vous (Pentium, PowerPC, ou autre), ne comprend pas le texte, compréhensible aux humains (du moins on l'espère) du fichier `cat.c`.

Pour savoir ce que le processeur comprend, on peut aller voir à quoi ressemble l'utilitaire `cat`, le vrai. D'abord, on cherche où il se trouve (c'est un fichier comme un autre !), en demandant `which cat`. Sur ma machine, la réponse est `/bin/cat`. Si je charge `/bin/cat` sous XEmacs, un éditeur de texte, j'obtiens quelque chose qui ressemble à :

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@
'\211^D^H4^@^@^@x!^@^@^@^@^@^@4^@ ^@^F^@(^@^X
^@^W^@^F^@^@^@4^@^@^@4\200^D^H4\200^D^HÀ^@^@^@
À^@^@^@^E^@^@^@^D^@^@^@^C^@^@^@ô^@^@^@ô\200^D
^Hô\200^D^H^S^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A
^@^@^@^@^@^@^@^@\200^D^H^@\200^D^H3^_^@^@3^_^@
^@^E^@^@^@^@^P^@^@^A^@^@^@4^_^@^@4¯^D^H4¯^D^Ht
^A^@^@,^B^@^@^F^@^@^@^@^P^@^@^B^@^@^@^H ^@^@^H
°^D^H^H°^D^H ^@^@^@ ^@^@^@^F^@^@^@^D^@^@^@^D^@
^@^@^H^A^@^@^H\201^D^H^H\201^D^H ^@^@^@ ^@^@^@
^D^@^@^@^D^@^@^@/lib/ld-linux.so.2^@^@^D^@^@^@
^P^@^@^@^A^@^@^@GNU^@^@^@^@^@^B^@^@^@^@^@^@^@
```

etc. Peu importe de ce que cela signifie exactement pour le moment, le point important est que, si ceci est compréhensible pour le processeur, c'est loin d'être lisible pour un être humain ! A contrario, le programme `cat.c` est bien plus lisible, mais le processeur ne le comprend pas.

La traduction du fichier `cat.c` (le *source*) en `cat` (l'*exécutable*) se fait au moyen d'un *compilateur*. Par exemple, la commande `gcc` est le compilateur C, et si l'on tape :

```
gcc -o mycat cat.c
```

le compilateur `gcc` va traduire (compiler) le source `cat.c` en un exécutable que j'ai décidé de nommer `mycat`, pour ne pas prêter à confusion avec l'utilitaire standard `cat`. Vous pouvez alors lancer `mycat`, et vérifier qu'il donne bien les résultats attendus en tapant la ligne de commande suivante, et en comparant avec ce qui se passait avec le programme `cat` :

```
./mycat a b
```

(Le "`./`" avant `mycat` nous sert à dire au shell que l'outil de nom `mycat` à utiliser est celui qui est dans notre répertoire courant.)

▸ **EXERCICE 1.1**
Que se passe-t-il si vous tapez la ligne suivante ?

```
./mycat cat.c
```

Vous pouvez aussi voir comment fonctionne `mycat` en utilisant le débogueur `ddd` :

▸ **EXERCICE 1.2**
Sous Unix, avec `ddd` installé (sinon, utilisez `gdb`, mais c'est moins lisible...) :

- Recompilez `mycat` en tapant `gcc -ggdb -o mycat cat.c`. L'option `-ggdb` permettra au débogueur de vous afficher des informations pertinentes (pour plus d'information, tapez `man gcc`).
- Tapez `ddd mycat &`. La fenêtre de `ddd` s'ouvre, montrant le texte source `cat.c`. Fermez la fenêtre "Tip of the Day" si besoin est.
- Cliquez sur le début de la ligne `int i, c;` avec le bouton droit de la souris, déroulez le menu qui s'affiche, et cliquez sur "set breakpoint". Une petite icône figurant un panneau stop rouge s'affiche par-dessus la ligne.
- En plaçant la souris sur la case du bas, où s'est affiché le message :
  ```
  (gdb) break cat.c:5
  Breakpoint 1 at 0x8048536: file cat.c, line 5.
  (gdb)
  ```
  tapez `run a b`. Ceci lance le programme `mycat`, avec arguments les chaînes de caractères `a` et `b`. Le programme s'arrête sitôt lancé sur le "breakpoint" que l'on a mis ci-dessus.
- Tirez le menu "View", et cliquez sur "Data Window" ; cliquez ensuite sur les menus "Data/Display Arguments" et "Data/Display Local Variables". Pour voir les arguments ("`a`", "`b`") qui ont été passés au programme `mycat`, cliquez sur le menu "Data/Memory...", et écrivez 3 en face de "Examine", puis "string" au lieu de "octal", enfin tapez `argv[0]`

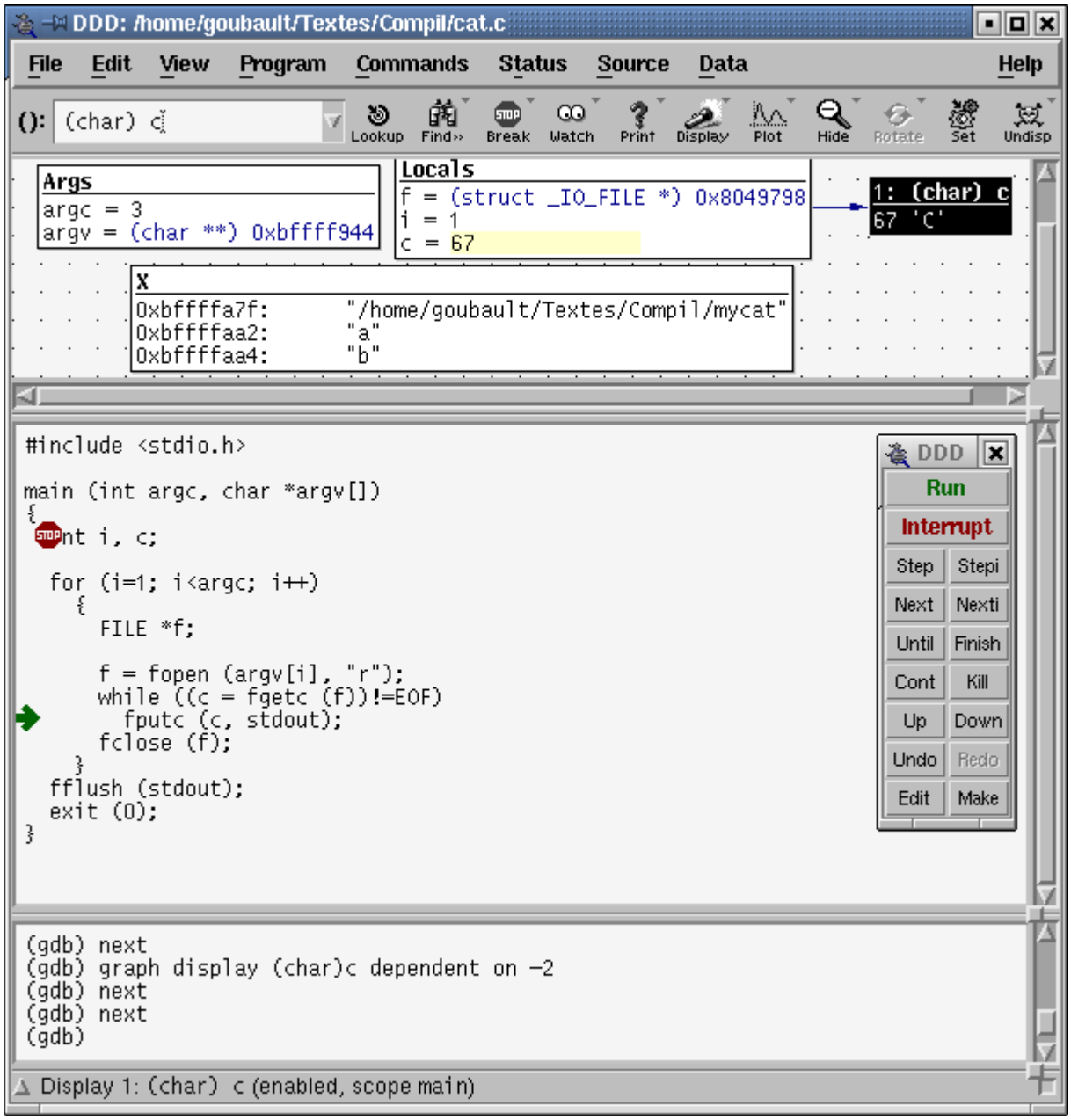

FIGURE 2 – Une session sous `ddd`

dans la case après “from”, puis cliquez sur les boutons “Display” puis “Close”. Vous aurez probablement besoin de réorganiser le cadre du haut de la fenêtre de `ddd` pour bien voir les affichages “Args”, “Locals” et “X”.

– Cliquez ensuite sur le bouton “Next” de la fenêtre flottante “DDD” répétitivement pour voir le programme s'exécuter petit à petit. Si vous voulez voir le contenu de la variable `c` sous forme d'un caractère lisible plutôt que sous forme d'un entier, cliquez avec le bouton droit sur la ligne de `c` dans l'affichage “Locals”, tirez le menu local “Display/Others...”, écrivez `(char)c` dans le cadre sous “Display Expression”, puis cliquez sur “Display”.

Vous devriez obtenir quelque chose qui ressemble à la figure 2 ; Vous pouvez bien sûr effectuer la même manipulation avec d'autres programmes, y compris ceux de factorielle ou de tri décrits plus hauts.

2 Langages impératifs, le langage C

Aujourd'hui nous examinons plus en détail les constructions de base d'un langage impératif typique : le langage C, inventé par Kernighan et Ritchie au début des années 1970.

La chose essentielle à retenir est qu'un programme C définit une *séquence* d'instructions élémentaires, qui sont des *affectations* : pour simplifier, les affectations sont des instructions de la forme ⟨variable⟩ = ⟨expression⟩`;`, qui ont pour effet de calculer la valeur de l'expression à droite du signe =, et de la ranger ensuite dans la variable à gauche du signe =. L'organisation des affectations en séquence se fait à l'aide d'une certain nombre de constructions du langage, que nous verrons en section 2.3. Cette présentation de C n'est pas complète, et on pourra consulter différents ouvrages de référénce, par exemple *Le langage C*, par Kernighan et Ritchie, ou la référence `http://diwww.epfl.ch/w3lsp/teaching/coursC/` par exemple. Pour les plus courageux, on pourra aussi consulter la norme ISO/IEC 9899-1999 définissant le C dit “ANSI C” (le C officiel).

2.1 Affectations

Par exemple,

```
x = 3;
```

a pour effet de mettre l'entier 3 dans la variable nommée `x`. Un autre exemple est l'instruction

```
x = y+1;
```

qui récupère l'entier stocké dans la variable `y`, lui ajoute 1, et range le résultat dans la variable `x`.

Attention ! Le symbole = n'est *pas* le prédicat d'égalité que l'on rencontre usuellement en mathématiques. La signification typique de $x = 3$ en mathématique serait celle d'une valeur de vérité, valant vrai si x vaut 3, et faux sinon. Ici, `x = 3` ne teste pas du tout si `x` égale 3 ou non. En revanche, il est vrai qu'une fois l'instruction `x = 3` exécutée, le contenu de la variable `x` vaut effectivement 3. Un moyen de voir à quel point = n'est pas le prédicat d'égalité des mathématiques est de considérer une instruction telle que :

```
x = x+1;
```

qui récupère l'entier stocké dans la variable x, lui ajoute 1, et range le résultat de nouveau dans x. Donc, par exemple, si x contient l'entier 3 avant d'effectuer `x = x+1;`, x contiendra 4 après. En particulier, en aucun cas on n'aura x égal à x+1 après exécution de cette instruction. Mais bien sûr la valeur de x après cette instruction égale la valeur de x avant.

Cette ambiguïté, qui peut surprendre ceux qui sont habitués aux mathématiques, a été levée dans d'autres langages impératifs, comme Pascal, où l'on écrirait `x := x + 1`, en utilisant le symbole := pour bien marquer qu'il s'agit d'une affectation et non d'un test d'égalité.

Si = est l'affectation, le test d'égalité en C est le symbol ==. On écrira donc :

```
if (x==3) ...
```

pour tester si x vaut 3, et non

```
if (x=3) ...
```

qui, dans un langage un peu plus sûr que C, serait une erreur, et serait rejeté par le compilateur. Mais C l'accepte gaiement... le langage C est en fait très permissif : la sémantique de C précise en effet que toute affectation, par exemple x=3, peut aussi être vue comme une expression, et a pour valeur la valeur de son côté droit. Ici, la valeur de x=3 est donc 3, ce qui permet d'écrire des expressions comme `y=x=3;`, qui range 3 dans x (x=3), puis retourne le résultat de l'affectation x=3, c'est-à-dire 3, et le range dans y : donc `y=x=3;` range 3 dans x et dans y.

La syntaxe des expressions de C est assez riche, et un aperçu (pas tout à fait complet, voir n'importe quel livre d'introduction à C pour l'ensemble complet) est donné en figure 3.

e ::=	`x` | `y` | `ma_variable` | ...	variables
|	`0` | `1` | `2` | ... | `-1` | `-2` | ... | `'a'` | `'b'` |	
	... | `3.1415926536` | ...	constantes
|	$e + e$ | $e - e$ | $e * e$ | e/e | e`%`e | $-e$	opérations arithmétiques
|	e`&`e | e`|`e | e`^`e | e`>>`e | e`<<`e | `~`e	opérations bit à bit
|	$e == e$ | $e < e$ | $e <= e$ | $e > e$ | $e >= e$	comparaisons
|	e`&&`e | e`||`e | `!`e	et, ou, non logiques
|	$*e$ | $e[e]$ | `&`e	pointeurs, tableaux
|	$e(e, \ldots, e)$	appel de fonction

FIGURE 3 – Un aperçu de la syntaxe des expressions de C

On ne décrira pas la sémantique des différentes expressions de C. Sur les entiers (qui sont les entiers machines, pas tous les entiers ! sur une architecture de machine 32 bits standard, les entiers sont ceux compris entre $-2^{31} = -2\ 147\ 483\ 648$ et $2^{31} - 1 = 2\ 147\ 483\ 647$), $e_1 + e_2$ calcule l'addition des valeurs de e_1 et e_2 (modulo 2^{32} sur une architecture 32 bits ; pour être précis, la valeur de $e_1 + e_2$ sur ces machines vaut l'unique entier compris entre -2^{31} et $2^{31} - 1$

qui est congru à la somme des valeurs de e_1 et e_2 modulo 2^{32}) ; $e_1 - e_2$ calcule la différence de e_1 et e_2 (modulo 2^{32} de nouveau sur une architecture 32 bits), $e_1 * e_2$ calcule le produit de e_1 et e_2 (modulo 2^{32}), e_1/e_2 calcule leur quotient et e_1`%`e_2 le reste de la division de e_1 par e_2 (modulo 2^{32}). Attention : e_1/e_2 ne calcule pas le résultat de la division de e_1 par e_2 (au type `int`... voir plus bas pour le cas des nombres flottants). A la place, e_1/e_2 calcule la partie entière de la division (le quotient, donc). Par exemple, $2/3$ en C donne 0, pas $0.666666\ldots$; et 2`%`3 calcule le reste, ici 2.

En revanche, on peut aussi calculer en C sur des *nombres flottants* (abréviation de "nombres à virgule flottante", qui dénote la façon de les représenter sur machine), qui sont des approximations finies de réels. Les opérations $+$, $-$, $*$ et $/$ sont alors les addition, soustraction, multiplication et division usuelles, à arrondi près. Par exemple, `2.0 / 3.0` donne vraiment `0.66666666666`. On remarquera que les nombres flottants sont écrits avec un point décimal ; en particulier, `2` et `2.0` sont deux objets différents : l'un est un entier machine, l'autre un flottant.

▸ **EXERCICE 2.1**

Que valent `1-2`, `15/7`, `15%7`, `1+(3*4)`, `(1+3)*4`, `1+3*4` en C ? Vous pouvez vous aider du compilateur `gcc`, en écrivant un programme de la forme :

```
#include <stdio.h>

int main ()
{
  printf ("La valeur de 1-2 est %d.\n", 1-2);
  return 0;
}
```

L'instruction `printf` ci-dessus a pour effet d'imprimer la valeur souhaitée : faire `man printf` pour avoir le modus operandi précis de `printf`.

Attention : ceci vous donnera la sémantique de `1-2` sur votre machine, pour votre compilateur particulier. N'en concluez pas nécessairement que la sémantique de C *en général* sera celle que vous aurez observée sur votre machine. Dans les exemples ci-dessus, cependant, les indications de votre machine seront fiables, et généralisables à toute machine (a priori).

▸ **EXERCICE 2.2**

Que valent `(-10)/7`, `(-10)%7` sur votre machine ? Ceci correspond-il à ce que vous attendiez ? Si q est le quotient et r le reste de la division de $a = -10$ par $b = 7$, a-t-on $a = bq + r$?

▸ **EXERCICE 2.3**

Que valent `10/(-7)`, `10%(-7)`, `(-10)/(-7)`, `(-10)%(-7)` sur votre machine ? Inférez-en la spécification du quotient et du reste sur votre machine : a/b et a`%`b calculent q et r tels que $a = bq + r$, $|r| < |b|$... et quelles autres propriétés déterminant q et r de façon unique ?

▸ **EXERCICE 2.4**

Il y a aussi toujours des cas pathologiques. Que valent `1/0`, `(-1)/0`, `0/0` sur votre machine ? (Attention, ceci est *totalement dépendant* de votre compilateur et de votre machine : sur la mienne, le programme est interrompu par un message `Floating point exception`, alors même qu'il ne s'agit pas de calcul en flottant mais bien en entier.)

▸ **EXERCICE 2.5**
(Si votre machine a des entiers 32 bits en complément à deux.) Rappelez-vous que le plus petit entier représentable en machine est $-2^{31} = -2\ 147\ 483\ 648$, et le plus grand est $2^{31} - 1 = 2\ 147\ 483\ 647$. L'opposé du plus petit entier représentable n'est donc pas représentable ! En expérimentant avec `gcc`, comment calcule-t-il l'opposé du plus petit entier représentable ? Que valent `(-2 147 483 648)/(-1)`, `(-2 147 483 648)%(-1)` ?

2.2 Tableaux, structures

J'ai un peu menti en section 2.1 en disant que les affectations étaient de la forme ⟨variable⟩= ⟨expression⟩ en C. Les côtés gauches peuvent en fait être plus généraux.

C'est notamment le cas avec les *tableaux*. En C, on peut déclarer un tableau, à l'entrée d'une fonction, par une déclaration de la forme

```
int a[50];
```

par exemple. Ceci déclare `a` comme étant un tableau de 50 éléments, chaque élément étant un entier machine (de type `int`). Alors que `int b` déclare `b` comme une variable contenant un seul entier, `a` ci-dessus en contient une rangée de cinquante.

On peut accéder à chaque élément du tableau en écrivant `a[`e`]`, où e est une expression retournant un entier. L'expression `a[0]` a pour valeur le premier élément du tableau, `a[1]` le deuxième, ..., `a[49]` le cinquantième (attention au décalage !). Les expressions `a[-1]`, `a[50]`, `a[314675]`, qui intuitivement dénoteraient des accès en-dehors du tableau, n'ont aucune sémantique : si l'on insiste pour obtenir leur valeur, on obtient en général n'importe quoi.

On peut aussi écrire des expressions d'accès aux éléments de tableaux plus compliqués. Par exemple, si `i` est une variable entière, `a[i]` dénote l'élément numéro `i` (qui est donc le `i`+1ième). L'intérêt de ceci est que l'on peut accéder à un élément variable, dont l'indice est lui-même calculé. Voici par exemple un calcul de produit scalaire de vecteurs à trois composantes :

```
float produit_scalaire (float a[3], float b[3])
{
  float resultat;
  int i;

  resultat = 0.0;
  for (i=0; i<3; i++)
    resultat = resultat + a[i]*b[i];
  return resultat;
}
```

(La boucle `for (i=0; i<3; i++)` itère l'instruction qui suit pour `i` allant de 0 inclus à 3 exclu.)

Pour changer le contenu des éléments d'un tableau, l'instruction d'affectation est en fait plus générale que celle que nous avons vue plus haut. On peut donc notamment écrire

```
a[0] = 5.0;
```

pour mettre le flottant `5.0` en premier élément du tableau (sans toucher aux autres). On peut ainsi effectuer une multiplication matricielle comme suit, par exemple :

```
  float a[3][3]; /* Multiplication de matrices 3x3, a et b,
                    résultat dans c. */
  float b[3][3]; /* A noter que les tableaux bidimensionnels
                    sont juste des tableaux à trois éléments
                    de tableaux à trois éléments. */
  float c[3][3];
  int i,j,k;

  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      {
        c[i][j] = 0.0;
        for (k=0; k<3; k++)
          c[i][j] = c[i][j] + a[i][k]*b[k][j];
      }
```

Les autres structures de données importantes en C sont les structures, les unions, et les pointeurs. Je ne parlerai pas des pointeurs ici, et je préfère attendre que vous ayez vu un peu d'architecture des machines ; les pointeurs ne se comprennent vraiment qu'une fois qu'on sait comment la machine fonctionne vraiment, à un niveau plus bas.

Les *structures*, appelées aussi *enregistrements* ("records") sont essentiellement des n-uplets. Par exemple, on peut définir le type des nombres complexes en C comme

```
struct complex {
  float re;
  float im;
};
```

Ceci définit un nombre complexe comme étant un couple de deux flottants (sa partie réelle et sa partie imaginaire). Étant donnée une variable complexe `z`, typiquement déclarée par :

```
struct complex z;
```

on pourra accéder à sa partie réelle en écrivant `x.re`, et à sa partie imaginaire par `x.im`. Les identificateurs `re` et `im` sont appelés les *champs* de la structure `z`. De même que pour les tableaux, on pourra affecter (la valeur d')une expression à chaque champ individuellement. Par exemple,

```
struct complex x, y, z;

z.re = x.re + y.re;
z.im = x.im + y.im;
```

calcule dans `z` la somme des deux nombres complexes `x` et `y`.

▸ **EXERCICE 2.6**
Écrire de même la différence, le produit, et la division de deux nombres complexes.

▸ **EXERCICE 2.7**
Écrire un programme de multiplication de deux matrices 3×3 de nombres complexes (à compléter) :

```
struct complex a[3][3];
struct complex b[3][3];
struct complex c[3][3];
int i, j, k;

for (i=0; ...
```

Les structures C permettent de décrire des *produits cartésiens* de types. Si A_1, ..., A_n sont des types C, le type `struct toto {` A_1 `a1;` ...A_n `an; }` est essentiellement un type de n-uplets d'objets pris dans A_1, ..., A_n.

On peut aussi décrire quelque chose qui ressemble à des unions de deux types :

```
union nombre {
  int i;
  float x;
};
```

par exemple décrit le type `union nombre` comme consistant en des objets qui sont des entiers machine ou bien des flottants. Je vous déconseille d'utiliser cette construction avant de bien comprendre ce qu'elle fait vraiment : consultez n'importe quel livre d'introduction à C. Attention, cette construction n'a en fait pas grand-chose à voir avec l'union ensembliste, malgré les apparences. Elle permet d'autre part toute une série de hacks (et toute une série de bogues aussi) que je ne peux décemment pas vous recommander.

2.3 Structures de contrôle

Pour organiser les affectations en séquence, le langage C, comme la plupart des autres langages impératifs, dispose d'un certain nombre de constructions de base, étendues par des formes dérivées (du *sucre syntaxique*) :

- l'instruction qui ne fait rien (!) : elle se note `;` , ou bien `{}` ;
- la *composition séquentielle*, ou *séquence* : c_1`;`c_2`;` exécute l'instruction c_1, puis e_2. Par exemple, `x=3; y=x+1;` commence par ranger 3 dans `x` ; une fois ceci fait, ceci calcule `x+1`, soit 4, et le range dans `y` ;
- le *test*, aussi appelé la *conditionnelle* :

`if (`e`)` c_1 `else` c_2

commence par calculer la valeur de l'expression `e`, qui est censée être un entier machine. Cet entier représente une condition booléenne (vrai/faux). Si cet entier est non nul (ce qui par convention signifie que la condition est vraie), alors c_1 est executée, sinon c_2 est exécutée. Par exemple,

```
if (x==3)
  printf ("x vaut bien 3.\n");
else printf("Ah, x ne vaut pas 3, tiens.\n");
```

À noter ici que la sémantique de `x==3` n'est pas exactement de retourner vrai si `x` contient l'entier 3, et faux sinon ; en fait, `x==3` retourne l'entier 1 si `x` vaut 3, et 0 sinon. D'autres langages impératifs ont un type spécial (`bool` en Pascal, par exemple) pour dénoter les valeurs de vérité, C non : en C, les booléens sont codés sous forme d'entiers.

– La boucle *while* :

$$\texttt{while } (e)\ c$$

commence par calculer e. Si e est vrai (i.e., un entier non nul), alors c est exécutée. À la différence de la conditionnelle `if`, une fois l'exécutioin de c terminée, la boucle revient au début, recalcule e : si e est vrai de nouveau, c est reexécuté, et ainsi de suite, jusqu'à temps que e devienne faux (le cas échéant).

On a déjà vu en leçon 1 que la boucle était sémantiquement une conditionnelle plus un point fixe :

$$\texttt{while } (e)\ c\ =\ \texttt{if } (e)\ \{c; (\texttt{while } (e)\ c)\ \}$$

Et c'est tout ! (Du moins, si l'on ignore les sous-programmes, dont nous parlerons plus tard.) On peut calculer tout ce qui est calculable avec ces seules instructions.

C propose aussi quelques extensions syntaxiques. Notamment, le *bloc*

$$\begin{array}{l}\{\\ \quad c_1;\\ \quad \ldots\\ \quad c_n;\\ \}\end{array}$$

permet de voir n'importe quelle composition d'instructions $c_1, \ldots, c_n$ comme si ce n'était qu'une (grosse) instruction. Ceci permet notamment d'écrire :

```
if (x==3)
  {
    y = z;
    if (z==4)
      printf ("Non seulement x vaut 3 mais z vaut 4.\n");
    else printf ("z ne vaut pas 4, mais y vaut z maintenant.\n");
  }
else
  printf ("x ne vaut pas 3.\n");
```

On peut aussi éviter d'écrire la partie `else` d'un test dans certains cas : `if (`e`)` c est une abréviation de `if (`e`)` c `else ;`.

Une autre construction très fréquente est la boucle *for* :

`for (`e_1`;`e_2`;`e_3`)` c

où e_1, e_2, e_3 sont des expressions (optionnelles : on peut ne rien écrire à la place de chacune), et c une instruction. Ceci est *exactement* équivalent à

e_1`;while (`e_2`) { `c`;`e_3`; }`

▸ **EXERCICE 2.8**

Réécrire la fonction `fact` du début de ce cours en terme de boucle *while*, comme indiqué ci-dessus. Ceci est-il lisible ?

▸ **EXERCICE 2.9**

L'idée de la construction `for` de C est de simuler une boucle pour `i` variant de m à n, comme la construction `for i=`a `upto` b `do`... du langage Pascal ou la construction `for i=`a `to` b `do ... done` de OCaml, en écrivant à la place `for (i=`a`; i<=`b`; i++) ...`. Cependant, en C, il est légal d'écrire :

```
for (i=1; i<=5; i++)
{
  printf ("La valeur de i est %d.\n", i);
  i = i+1;
}
```

Notez que la bizarrerie est que l'on change la valeur de `i` dans le corps de la boucle une seconde fois, en avant-dernière ligne. Que fait ce bout de code ? Peut-on faire pareil en OCaml ?

Finalement, en C on a aussi une construction `switch`. Disons en première approche que le code :

```
switch (e) {
      case ⟨constante 1⟩: c1; break;
      case ⟨constante 2⟩: c2; break;
      ...
      case ⟨constante n⟩: cn; break;
      default: c; break;
```

fait la même chose que (où `i` est une variable fraîche) :

```
{
 int i;
 i = e;
 if (i==⟨constante 1⟩)
    c1;
```

```
else if (i==⟨constante 2⟩)
        c_2 ;
else if ...
   ...
else if (i==⟨constante n⟩)
        c_n ;
else c;
```

La construction `switch` est censée être plus rapide que la suite de `if` ci-dessus. Elle admet aussi quelques variantes, notamment, si l'on omet le mot-clé `break` en fin d'une ligne `case`⟨constante i⟩, l'exécution de c_i se continuera sur celle de c_{i+1}. C'est une source courante d'erreurs en C, et un des charmes particuliers du langage.

3 Représentation des données I

En regardant le programme de concaténation de la figure 1, on voit des objets `i` et `c` de type `int`. Commençons par examiner ce type de données. Nous éluciderons les types plus complexes `char *` et `FILE *` plus tard.

Le type `int` est le type des *entiers machine*. Tous les entiers mathématiques ne sont pas représentés tels quels sur une machine, et les ordinateurs travaillent plus rapidement sur des entiers dans un intervalle fini, les entiers machine.

Il faut d'abord comprendre qu'un ordinateur ne stocke réellement que des collections de *bits*, c'est-à-dire des booléens, 0 ou 1. Les ordinateurs regroupent traditionnellement les bits par groupe de huit : une suite de huit *bits* s'appelle un *octet* (*byte* en anglais). Le type des octets est `char` en C... car ils servent traditionnellement à coder les caractères, mais n'anticipons pas. En attendant, il est pratique d'imaginer un octet comme un nombre à 8 chiffres en base 2. Par exemple, la suite de 8 bits 0000 1101 est le nombre $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$—de même qu'en base 10, le nombre 451 est $4 \times 10^2 + 5 \times 10^1 + 1 \times 10^0$.

La *mémoire* d'un ordinateur est ensuite organisée comme un gigantesque tableau d'octets. Les indices de ce tableau sont des nombres, typiquement de 0 à $2^{32} - 1 = 4\ 294\ 967\ 295$ sur les ordinateurs dits 32 bits. Les indices de ce tableau sont traditionnellement appelés les *adresses*, et l'on voit qu'on peut représenter toutes les adresses par des suites de 32 bits (d'où le nom d' »ordinateur 32 bits »), sur le même principe que les octets.

On peut voir en figure 4 un exemple de mémoire d'un ordinateur. Comme le montre le tableau du haut de la figure 4, une mémoire n'est donc rien d'autre qu'une fonction qui à chaque adresse associe un contenu, qui est un octet. Il est traditionnel de compter les adresses et de lire les octets en *hexadécimal*, c'est-à-dire en base 16 ; la lettre `a` vaut 10, `b` vaut 11, ..., `f` vaut 15. Ainsi les adresses sont 0, 1, ..., 8, 9, a, b, c, d, e, f, 10, 11, ..., $1a$, $1b$, ..., $1f$, 20, ... Lorsqu'il y aura ambiguïté dans la base, on utilisera la convention du langage C de faire précéder une suite de caractères parmi `0`, ..., `9`, `a`, ..., `f` des caractères `0x` pour montrer que le rester est un nombre hexadécimal. Par exemple, `0x10` dénote 16 ; ou bien `0xcafe` vaut $12 \times 256^3 + 10 \times 256^2 + 15 \times 256 + 14 = 51\ 966$.

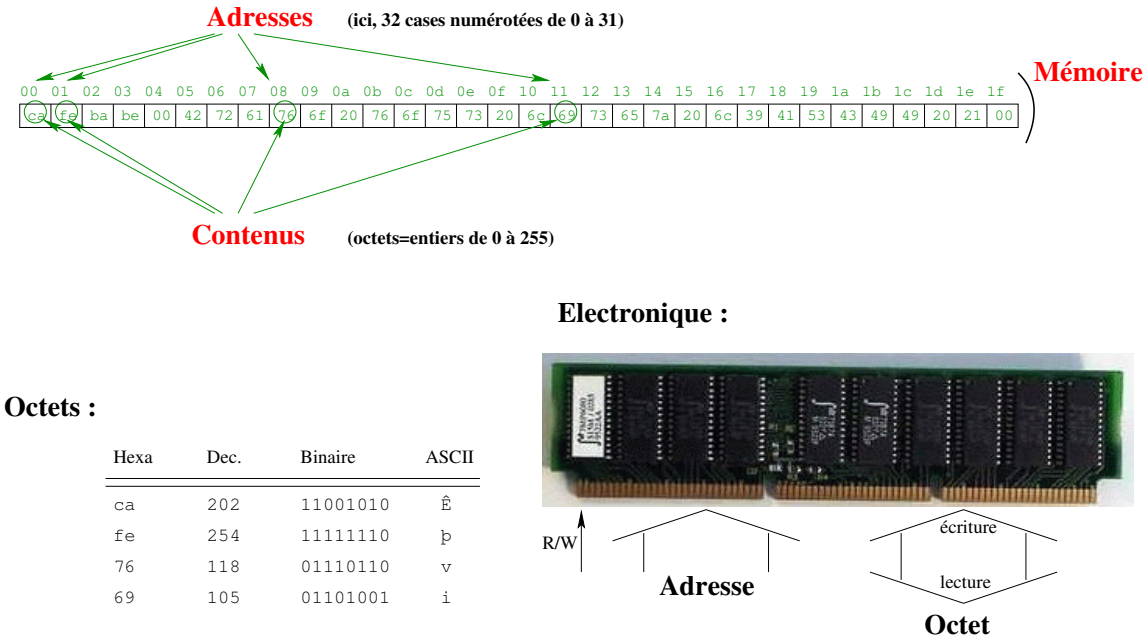

FIGURE 4 – La mémoire des ordinateurs

De même, les octets seront donc les nombres de `0x00` à `0xff`.

La mémoire du haut de la figure 4 contient donc l'octet `0xca`$= 12 \times 16 + 10 = 202$ à l'adresse `0x0`$= 0$, l'octet `0xfe`$= 15 \times 16 + 14 = 254$ à l'adresse `0x1`$= 1$, et ainsi de suite.

Électriquement, une mémoire est un circuit ou un groupe de circuits ressemblant à celui en bas à droite de la figure 4. Les adresses sont envoyées sous forme de signaux électriques (par exemple, si le bit i de l'adresse vaut 1 alors on relie le contact électrique no. i au +5V, sinon à la masse). Un autre contact, `R/W`, est positionné à 1 si l'on souhaite lire l'adresse ainsi décrite, auquel cas le contenu de l'octet à cette adresse sera présent en binaire sur huit autres contacts. Si `R/W` est positionné à 0, alors le circuit de mémoire s'attend en revanche à ce que la donnée à inscrire à l'adresse précisée soit déjà présente sur les huit contacts ci-dessus. Ceci est le principe général, il y a en fait d'autres contacts, et d'autres conventions (par exemple, le +5V peut en fait coder le 0 au lieu du 1, la tension peut ne pas être +5V, etc.)

Il est d'autre part important que tous les octets sont utilisés comme *codages*. L'octet `0xca` peut être vu comme :

- l'entier positif ou nul $12 \times 16 + 10 = 202$, comme ci-dessus (on dit que c'est un entier *non signé* ;
- ou bien l'entier compris entre $-2^7 = -128$ et $2^7 - 1 = 127$, et égal au précédent modulo $2^8 = 256$: ceci permet de représenter des entiers positifs ou négatifs dans un certain intervalle ; selon cette convention, `0xca` vaut alors $202 - 256 = -54$, et l'on parle d'*entier signé en complément à deux* ;
- ou bien le code du caractère "Ê" : il s'agit d'une pure convention, que l'on appelle la table ASCII (American Standard Code for Information Interchange), ou plutôt la table ISO-8859, voir la figure 5 (source : `http://www.bbsinc.com/iso8859.html`).

Par exemple, le caractère de code `0x43` se trouve dans la ligne `40`, colonne `3` : il s'agit du `C` ;

– ou bien l'instruction assembleur `lret` (“long return”) du processeur Pentium ;
– ou bien encore pas mal d'autres possibilités. . .

Tout dépend de ce que l'on fait avec l'octet en question : des opérations arithmétiques sur entiers signés, sur entiers non signés, des opérations de manipulations de chaînes de caractères, exécuter un programme, etc.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
80			‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ			
90		‘	’	“	”	•	–	—	˜	™	š	›	œ			Ÿ
A0		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

FIGURE 5 – La table ASCII étendue

3.1 Entiers machines

Concentrons-nous sur les entiers machines pour commencer.

Pour diverses tailles (8 bits pour les octets, 32 bits pour les entiers machine sur les machines dites 32 bits, 64 bits sur les machines modernes), disons de n bits, les *entiers non signés sur* n

bits sont juste les suites $b_{n-1}b_{n-2}\dots b_1b_0$ de n bits bien sûr. Ce qui fait qu'on les voie comme des entiers *non signés*, c'est notre décision qu'une telle suite représente le nombre $b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_1 2^1 + b_0 2^0$.

Le type des entiers machine non signés (32 ou 64 bits typiquement) est `unsigned int` en C. Le types des octets non signés est `unsigned char`. L'addition, la multiplication, et toutes les opérations arithmétiques sont interprétées modulo 2^n, où n est le nombre de bits.

Une conséquence est la possibilité de *débordement arithmétique*. Par exemple, sur des octets non signés, l'évaluation de $200 + 200$ ne donnera pas 400, qui n'est pas représentable, mais le reste de 400 modulo $256 = 2^8$, soit 144. L'existence de débordements arithmétiques mène à de nombreux bogues, liés à l'illusion que l'on calcule sur de vrais entiers.

Comment représente-t-on des entiers relatifs, c'est-à-dire qui peuvent être positifs ou négatifs ?

Une ancienne façon de faire est le *complément à 1*. Sur n bits, on réserve un bit de signe (0 pour positif, 1 pour négatif), et les $n-1$ autres représentent la valeur absolue du nombre. Ceci a l'avantage de permettre le calcul de l'opposé très simplement, en inversant juste le bit de signe. Mais cette représentation souffre de bizarreries. Par exemple, il existe *deux* représentations du nombre 0, notées traditionnellement $+0$ et -0, en exhibant le bit de signe explicitement.

La plupart des ordinateurs modernes, et ce depuis longtemps, utilisent plutôt le *complément à 2*, dont nous avons déjà parlé un peu plus où. Les *entiers signés sur n bits en complément à deux* sont de nouveau juste les suites $b_{n-1}b_{n-2}\dots b_1b_0$ de n bits. Ce qui fait qu'on les voie comme des entiers *signés*, en complément à deux, c'est notre décision qu'une telle suite représente le nombre (positif ou nul) $b_{n-2}2^{n-2} + \cdots + b_1 2^1 + b_0 2^0$ si $b_{n-1} = 0$, et le nombre (négatif) $-2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_1 2^1 + b_0 2^0$ si $b_{n-1} = 1$.

Par exemple, sur 8 bits, les *octets signés* (c'est le type `signed char` de C… le type `char` est signé ou non, au gré du compilateur !) sont la suite de bits $0000\ 0000$ représentant 0, $0000\ 0001$ représentant 1, …, $0111\ 1111$ représentant $2^7 - 1 = 127$; puis $1000\ 0000$ représentant $-2^7 = -128$, $1000\ 0001$ représentant -127, …, $1111\ 1110$ représentant -2, $1111\ 1111$ représentant -1.

La bonne nouvelle, c'est que le premier bit b_{n-1} (aussi appelé *bit de poids fort*, le *bit de poids faible* étant b_0) est toujours le signe, comme en complément à 1 !

La mauvaise, c'est que la représentation semble un peu plus obscure. Mais tout devient plus simple ! L'addition non signée par exemple reste valide dans le cas signé. En fait, l'arithmétique signée en complément à deux reste, comme l'arithmétique non signée, juste la version modulo 2^n de l'arithmétique exact. Au lieu de prendre le reste modulo 2^n, c'est-à-dire l'unique entier compris entre 0 inclus et 2^n exclu égal au résultat exact, on prend l'unique entier compris entre -2^{n-1} exclu et $+2^{n-1}$ inclus égal au résultat exact.

3.2 Caractères

J'ai menti en disant que la table 5 était la table ASCII. Ce ne sont que les 128 premiers caractères (du haut) qui forment la table ASCII : A est pour « American », et les américains n'ont aucun besoin de caractères accentués ou encore plus exotiques.

Lorsque les constructeurs informatiques ont eu besoin, dans les années 1980, de permettre aux utilisateurs d'écrire des textes avec d'autres caractères, ils ont pensé utiliser les 128 autres ca-

ractères disponibles dans une représentation par octet. Ceci a mené par exemple à la table 5. Mais les ordinateurs Mac et les ordinateurs sous Windows avaient défini des codages de caractères accentués incompatibles... de plus, il n'y avait aucun moyen de loger dans une même table de 256 caractères possibles tous les caractères accentués, les ligatures, les caractères spéciaux, de ne serait-ce que les langages européens.

Le véritable changement est advenu dans les années 1990, où le standard Unicode a commencé à s'imposer. On y code notamment tous les caractères accentués, les 29 lettres de l'arabe, celles de l'hébreu, les idéogrammes chinois, et les quatre alphabets japonais notamment.

Le prix à payer est que chaque caractère nécessite bien plus de 8 bits. Le codage de base de l'Unicode, appelé UTF-32, code chaque caractère sur 32 bits. Mais il faut bien dire que c'est un peu lourd. Le codage le plus répandu aujourd'hui est l'UTF-8 (imposé dans le langage Java notamment), qui code les caractères sur un nombre variable de 1 à 4 caractères. Le principe est le suivant. On lit le premier octet d'un caractère. Si le bit de poids fort est à 0, alors le caractère tient en cet unique octet, et ce caractère est donné par la table ASCII usuelle. Sinon, il faut lire le deuxième octet. Cet octet donne des indications sur la table qui va servir à décoder le tout, et sur le nombre total d'octets à lire.